DOCUMENT PREPARATION SYSTEMS
J. Nievergelt, G. Coray, J.D. Nicoud, A.C. Shaw (eds.)
© North-Holland Publishing Company, 1982

ERRORS IN DIALOG DESIGN AND HOW TO AVOID THEM

J. Nievergelt

Institut für Informatik, ETH, CH-8092 Zurich, Switzerland

Abstract

The development of interactive command languages is repeating, with a 20-year delay, the history of programming languages: large collections of unrelated individual commands are being replaced by systematically structured modes that satisfy general principles of consistency. A survey and classification of design errors common in today's man-machine dialogs leads to concepts and design principles whose observance avoids such errors.

But the very notion of *command language* is of doubtful utility for systematizing the design of highly interactive man-machine interfaces. Observation of users of interactive systems shows that the most common difficulties experienced are expressed well by such questions as: *Where am I? What can I do here? How did I get here? Where else can I go and how do I get there?* Such questions reveal the fact that the fundamental problem of man-machine communication is how to present to the user what he cannot see directly, namely the state of the system, in such a manner that he can comprehend it at a glance. If the *output language* of a system, what the user sees or can get onto the screen easily, is rich and well structured, the command language proper, or *input language*, can be simple. Instead of emphasizing the *command part* of the man-machine interface, designers should focus their attention on the *display part.*

The questions above also provide hints about how the display of an interactive system should be structured. *Where am I?* and *What can I do here?* refer to the state of the system. A good design answers them by displaying to the user on demand his current data environment *(site)* and his current command environment *(mode).* The last two questions refer to the past and future dialog *(trail).* A good design makes as much of the past dialog as can be stored available to the user, to be undone (in case of error), edited and reinvoked; and it gives advice about possible extrapolations of the dialog into the future *(help).*

Key words and phrases

Interactive systems, man-machine communication, system design

Contents

1. Anecdotes, horror stories, and what we can learn from them
2. Survey and classification of errors
2.1 Insufficient state information
2.2 Poor design of the interface
2.3 Ignoring the user's trail
2.4 Compounding the problems
3. An interactive system as seen by the user
4. Programming the man-machine interface

1. Anecdotes, horror stories, and what we can learn from them

Recently I concatenated several files that contained sections of a report into a single file. Bitter experience has taught me that file operations on many a system are error-prone. With relief I saw the target file name appear in the directory; but when I wanted to look at the result, the editor complained: file too long! In interactive systems a text editor is the primary utility for splitting and merging files, since it allows the user to browse through the file and exploit his eyes' pattern recognition ability. Deprived of my most powerful tool, I realized that this "correct" file that could not be seen was useless. Only half in jest I said: "I bet the file will be too long to be deleted". The quickest remedy was to start from scratch, catenating as many of the original files as would fit.

So what? Today's programmer accepts as a matter of course the fact that, on the same system, an editor, a file server, a printing routine all have their own file size limits.

A few months ago, as I was editing at an "intelligent" terminal, my friend inadvertently switched the power off the minicomputer to which my terminal was attached. There was no way I could have noticed from the terminal's behavior that I was off-line, so I continued editing for half an hour before attempting to write the buffer content onto a power-off disk. Starting up the system re-initializes the terminal, so there was no practical way to save the file content that was still visible on the screen. The most efficient remedy was to scribble the major modifications onto paper and to redo them.

With foresight this system could have been designed so that the nuisance above could never happen. Why was it not done? Today's computer user routinely wastes hours with boring work just to get around trivial design flaws.

A couple of years ago a competent computer man grabbed the nearest disk pack in a hurry to run a test program. The program required a large block of contiguous disk space, so he applied a *squeeze operation* to the disk content. The drive misbehaved, the directory of my disk got destroyed. He was choking as he called late at night to announce the loss of my files. The remedy? He spent several days writing a *scavenger*, a program that searches the disk trying to identify file headers from which the file structure can be reconstructed. We spent another day "manually" pasting file fragments together, were able to reconstruct most of the old disk content that mattered, and clean up the disk as a beneficial side effect. The scavenger joined our collection of utilities.

What should he and I have done to avoid this near-disaster? I hear a sermon: "Obviously you use a scratch disk for running hardware tests and keep up-to-date backups!" True. Similar advice is given to rock climbers. But notice that we don't use a scratch record to try out a record player, and that few people keep a backup car. Now that computers are becoming consumer products, they have to start behaving accordingly. Anecdotes and horror stories such as the ones above, instead of describing the state-of-the-art, must become tales of the good old days.

The spell of an anecdote lies in the unexpected; the reliability of a system manifests itself in that only the expected happens. Suspense and reliability are incompatible: We must learn to build systems that never catch the user by surprise. The history of mature fields of endeavor teaches us how to achieve this goal: we must suffer a lot of surprises, study and classify these, develop concepts and principles that exclude the known errors, and enforce them. No other way can relegate painful surprises from everyday experience to rare accidents.

Over the past decade I have observed hundreds of users of interactive systems suffer many surprises in a variety of settings. I have analyzed these impasses, tried to reduce them to a small number of causes, and proposed design techniques to avoid them. The procedure used is common sense observation and deduction. A similar approach to developing design rules based on the analysis of human errors is described in [Nor 82]. More rigorous techniques have recently become fashionable, and the literature on the psychology of human-computer interaction has mushroomed. [Mor 81] is a good introduction to this subject. But the greatest improvements on the sad current state of man-machine dialogs can be achieved with common sense alone, without the fine-tuning of controlled experiments. The following sections are intended as a casual introduction.

2. Survey and classification of errors

All roads lead to Rome. In a thicket of superficially unrelated phenomena, all considerations about man-machine dialogs lead to two fundamental concepts: *the STATE of the system and its INTERFACE to the user.*

The *state* is concerned with everything that influences the system's reaction to user inputs. The *interface* is concerned with interaction: what the user sees on the screen, what he hears, how he inputs commands on the key board, mouse or joy stick. Today's fad is to engineer fancy interfaces: fast animation in color, sound output and voice input. This is desirable for some applications, unnecessary for others, but in any case it does not attack the main problem of man-machine communication.

The fundamental problem of man-machine communication is how to present to the user what he cannot see directly, namely the state of the system, in such a manner that he can comprehend it at a glance.

So let us start with common errors of dialog design which hide vital state information from the user.

2.1 Insufficient state information

Imagine that you leave your terminal in the middle of an editing session because of an urgent phone call. When you return ten minutes later, the screen looks exactly as you left it. Even if the system state is unchanged, you may well be unable to resume work where you left off:

D: "What file was I editing?", "Is there anything useful in the text buffer?";

C: "Am I now in search mode?", "What is the syntax of the FIND command?";

T: "Has this file been updated on disk?", "Has it been compiled ?".

Such questions indicate that part of the state information necessary to operate this system has to be kept in the user's short term memory. When the latter is erased by a minor distraction, the user needs to query the system to determine its state. Hardly any system lets him do this systematically, at all times.

Today's systems provide state information sporadically, whenever the programmer happened to think about it. The file directory can be seen in the file server mode, but not in the editor; in order to inspect it you must exit from the editor, an operation that may have irreversible consequences. In order to see the content of the text buffer you may have to insert it into the main text. In order to find out whether you are in search mode you may have to press a few keys and observe the system's response - not always a harmless experiment.

A designer who follows the following principle avoids all the problems above:

The user must be able AT ALL TIMES to conveniently determine the entire STATE OF THE SYSTEM, WITHOUT CHANGING THIS STATE.

The phrase "entire state" will become more precise in section 3, but one refinement of this notion can be explained now. In any interactive system the user operates on data (perhaps he only looks at it) by entering commands (perhaps he only answers multiple-choice questions). Thus the state of the system must include the following two major components: the *current data environment* (what data is affected by commands entered at this moment), and the *current command environment* (what commands are active). The questions above labelled D and C refer to data and command environments, respectively. The questions labelled T refer to the user's *trail*, i. e. the dialog he is conducting during a session.

A particularly dangerous version of "insufficient state information" is the deceitful presence on the screen of outdated state information. The programmer is consciously aware of the moment he has to write some information on the screen, but he forgets to erase it when it no longer describes the state of the system. The user sees 'current file is TEMP' in some corner of the screen, when in fact his current data environment is another file. The bad habit of leaving junk lie around the screen must have originated in the days of the teletype, when messages written could not be erased, but it should not be allowed to persist on today's displays.

So our answer to the question: *Is it really necessary for an interactive system to have so many commands?* is: not if the general dialog commands are shared by all the interactive utilities on the system, and the latter limit themselves to the commands that are specific to the type of data they handle. The solution to both problems of too many commands and of the Tower of Babel confusion lies in the concept of **universal commands**. An interactive system should have a set of about a dozen general dialog commands that are universal in several meanings of this word:

they are always active,
they are always invoked by the same key presses, and
they always have the same interpretation regardless of the type of objects currently being handled.

Commands are the ingredients of the system's input language; responses make up the system's output language. Because the user actively composes commands, but only sees or hears responses "passively", it is perhaps natural that initially the "command part" of a command language has been stressed. With terminals of limited display power as those traditionally used, it was indeed a necessity to code most of the complexity of the man-machine interface into the input part of a command language. Now that wide-bandwidth graphics terminals are becoming common, it is a mistake. The complexity of the interface is much more effectively embedded in the *response part* of a command language, and the input part should approximate as much as possible the command language with one single command, pointing. For this to become possible, a designer must impose a rich and clean structure on the output language, so that the user can easily explore the state of the system (see [XS 82} for an example).

An elementary but common violation of the duty to pay attention to the output language is omission of feedback to the user. You enter a command and wait; if the response is not instantaneous, you may start wondering: have I hit return already, or is the system waiting for me to do so and thus activate the command? is it possible that a squeeze operation takes this long? has the system crashed? Without feedback the user has no good option in this situation: if he presses return twice, the second key press may be interpreted as answer to the next system prompt; if he aborts the command he may do irreparable damage; if he decides to wait, how long should he wait?

The reason that today's dialogs provide feedback only sporadically is the same one that causes screens to be littered with outdated information. The programmer of a man-machine dialog tends to identify himself with the machine - after all, he communicates with the user via the medium computer-driven screen [Nie 82]. Thus those moments when he (or the computer) has to take the initiative are foremost in his mind, such as when he has to put a menu on the screen; but he forgets to reassure the user when it is the latter's turn to be active .

The only way to avoid leaving the user in suspense is to follow a systematic procedure of providing feedback as a default, and to suppress it only by conscious decision. Feedback should be provided as a matter of course on at least the following four levels (see also [Fol 80]):

- the key press level (usually by echoing)
- the lexical level (e.g. a parameter of type *integer* has been recognized)
- the syntactic level (e.g. a command has been recognized)
- the semantic level (e.g. a command is being processed or has been completed).

A beep that announces completion, or the continuing appearance of dots on the screen to inform the user that the system is up and working on his problem, can be amazingly reassuring. The programmer just has to place the feedback where it will be seen, usually near the user's last input, or else in a standard place on the screen to which the user has become accustomed. But this advice already belongs to our next topic, that of:

Encoding of commands and system responses

Is the "/*$ syntax" of many of today's command languages the fault of the limited character sets of early hardware or of the indolence of the programmer who wrote the command interpreter? Whatever the reason, '/*C $X $Y' is of little mnemonic value. 'Copy from .. to ..' makes everything clear, in particular the important issue whether the first or the second argument is the destination to be overwritten.

Undiplomatic and uninformative encodings of system responses are common. 'Illegal command' is one of the most frequent messages to appear on the screen. The appearance of a menu or other information that

2.2 Poor design of the interface

The interface, that is, the face that a system presents to the user, has a ***physiological*** and a ***logical*** component. The former is the traditional subject of ergonomics or human factors engineering. Many frequent violations of established ergonomic principles can be cited: rigidly fixed key boards that force the user into an unnatural position; flickering displays; excessive use of inverse video techniques that produces stark contrasts in brightness among different parts of the display; placing the display in front of a window, thus forcing the user to look into a surrounding area much brighter than the screen. The eyes tire as they continuously adapt in their sweep of the screen and its surroundings; [GV 80] discusses such issues. I will concentrate on the neglected topic of the ***logical design of the man-machine interface.*** This discussion is best separated into two subtopics:

- the structure of the command language, and
- the encoding of commands and responses.

Structure of the command language

A command language consists of two parts, an input language (the set of commands) and an output language (the system's responses). The former is traditionally considered to be the important part, hence the name "command language", in analogy to "programming language". We will argue that "command language" is a misnomer, that in fact the output part of the language is the important thing, but in order to mirror the development of the field we discuss the input part first. Interactive systems have a lot of commands. It is not easy to count them, since one single command with many parameters can always be traded for many simple commands. But if one counts a command with the multiplicity of the number of parameters it has, that is, with the number of distinct decisions that the user may have to make, one easily reaches a hundred. As an example, a glance at the manual of a successful interactive system for hobby computers (UCSD Pascal [AP 79]) yields the following approximate counts: *Top command level: 8 commands without parameters. Filer: 18 commands with an average of one or two parameters. Editor: 28 with an average of one or two parameters.* Even without counting 20 compiler options and an untold number of error messages that the system may throw at the user, we see that even a small system requires the user to have present in his mind many more chunks of information than he can keep in short term memory (see "The magical number seven, plus or minus two" [Mil 56]). Systems for professional use (CAD, document preparation, etc.) are even more complex.

The consequence of this complexity is that the system can only be used effectively by those who have memorized this large set of commands, not by casual users who must retain the information necessary for working in their short-term memory, which is subject to the limitation of half a dozen chunks at a time [Mil 56]. And the cost of memorizaton is high: It takes me at least a week to get to know an interactive system well enough to work at my level of capacity; if I stop using it for a month, it may take a day to regain the proficiency I had earlier.

Is it really necessary for an interactive system to have so many commands? An analysis of various interactive utilities, such as editors for different types of objects (text, diagrams, forms) yields a surprising answer: only a minority of the commands are specific to the type of objects to be manipulated by each editor; the majority of the commands have a general interpretation that is independent of whether we speak of text lines or graphic lines, paragraphs or pictures. There are *dialog control commands* such as next, back, quit that are completely object-independent; general *utility commands* such as copy, create and delete that only depend on the fact that an object can be uniquely identified, say by naming or by pointing; there are *motion commands* such as up, down, left, right that only depend on an ordering of the objects, and can be used equally well to move from line to line, file to file, or picture to picture.

It is the major shortcoming of today's interactive systems that every interactive utility introduces another set of commands necessary to conduct a general dialog, rather than limiting itself to the data-specific commands necessary to manipulate its own type of objects. This leads to the sad state of affairs that the programmer of each utility re-invents his own general dialog commands. Quit (from the current command or data environment) will always be present, but it will be named Q in one utility, E or X (for exit) in another, and be invoked by Control-C or the escape key in a third. This multiplicity of names may leave the user in doubt whether the state of the system is different after executing differently named quit commands. The casual user in particular bears the burden of this Babylonian language confusion.

tells the user just what inputs are expected at that moment is less offensive and a lot more helpful. The kindest statement possible about 'Illegal command' is that it beats a system crash - a cruel but regrettably not unusual punishment for "illegal inputs".

There are manuals of style for speaking and writing, and we need manuals of style for novice authors of man-machine dialogs. Proper expression is not just a matter of esthetics; it is also a matter of efficient use of people's time. [LWSS 80] presents the results of an experiment designed to investigate the effect of ill-chosen names and syntax on user productivity. Two groups of users ranging from novices to experts were given the same editing tasks. They also used the "same" text editor - except that the commands had different names and a slightly different syntax. Whereas the original editor conversed in phrases such as 'FIND:-5' and 'RS:/KO/,/OK/', the revised editor called the same two commands 'backward 5 lines' and 'change "KO" to "OK"', whose meaning can be guessed without a manual. As you would expect, experts as well as novices did better with the editor using meaningful names with respect to work completed and number of errors.

2.3 Ignoring the user's trail

"We have seen the enemy, and he is us!" The user is by far the most dangerous component of an interactive system - at least 90% of all mishaps that occur during operation are traceable to faulty user actions. Designers of interactive systems must accept the fact that high interactivity encourages trial-and-error behavior. Exhortations to discipline are worse than useless - they are counter-productive. A designer using a CAD system, a writer using a text processor, a programmer debugging his programs - they must concentrate on their creative task, and cannot allow a fraction of their conscious attention to be sidetracked, continuously double-checking clerical actions. A system should be foolproof enough to absorb most inaccuracies and render them harmless.

It is surprising that virtually all research on reliability and security is directed against either hardware and software failure or deliberate attack by third parties. But the majority of users of interactive systems have no one but themselves to blame when their data is suspect or has been damaged. In cleaning up one's files the most recent version of a document is thrown away and an old one kept instead. You hit **D** instead of **C** on the keyboard, so something gets deleted instead of created. You forget to label a file permanent and so it vanishes. The catalog of plausible errors is different on each system; they all have in common that, as soon as we become aware of the blunder, we gasp: "How could I possibly have done that!". We can and do, at the rate of many oversights a day.

The mechanisms built into today's systems to protect the user from his own mistakes are primitive. Are you sure? is one of the favorite questions, followed by a sporadic request to press some unusual key if the action is really serious. This double-checking is effective only against accidental key pressing, not against an erroneous state of mind. When I delete a file I'm sure that's what I want to do - though I may regret it later.

The most effective protection of the user from his own mistakes is to store as much of the past dialog as is feasible. At the very least a universal command undo must always be active that cancels the most recent command executed. The memory overhead of keeping two consecutive system states is negligible, since these two states differ little - typically by at most one file. The error-prone and time-consuming Are you sure? is unnecessary when the user can always undo the last step. Ideally undo works all the way back to the beginning of the session, but that may be too costly: in order to replay the user's trail *backwards* the system must in general store *states*, not just the *commands* entered, as many operations have no inverses (e.g. delete). A practical alternative is for the system to keep a log of commands entered and to allow the user to save the current state as a check point for future use, from which he can replay the session.

2.4 Compounding the problems

Every one of the design errors mentioned above may seem mild when considered in isolation. While reading section *2.1 Insufficient state information* you expect to face errors of that kind, and hence you recognize them. But in an unsuspecting state of mind the user is likely to take every misleading system message at face value. I remember a user walking away from a time-sharing terminal muttering "the system is down again". "It's up", I said. "No, it doesn't react to any of my commands". He had somehow gotten into the editor's insert mode, but the indication of this mode change had long ago been scrolled off the

screen. *He* thought he was entering commands, whereas *the system* thought he was entering text, and that the only thing it was supposed to do was to faithfully echo his key presses, just as the terminal does in local mode.

In "Breaking the man-machine communication barrier" [HBR 81], an amusingly sad story is told of how intolerant a system can be. The user of an electronic mail system asks to see all the messages that have recently arrived from a particular sender. He types a long command: 'haeders from robertson since May 15', which contains minor syntactic errors. The system tolerates no misspelling and has no memory of the previous command line entered by the user; it rejects his attempts time and again with uninformative comments such as '?illegal message sequence'. Trial and error and help from expert users finally lead to the correct formulation: 'headers from "robertson" intersect since "May 15"'.

The barrier illustrated by the example above is caused not by any single design error, but by the cumulation of imperfections that result in the syntactically correct form of the command being intuitively less meaningful than the original faulty formulation. Several improvements should occur immediately to the skilled designer of man-machine dialogs, such as:

A casual user needs a mode that prompts him for selection of one among a few options, or for input of a single parameter; the *user control mode* exhibited in the example above, where a long command with several parameters must be entered without guidance, is only appropriate for skilled users.

In limited domains of discourse (and programs are only able to converse in very limited domains of discourse) the system can afford to be tolerant about the precise form in which a command or parameter is entered. A date, for example, should be accepted in any of the forms it is usually written. In order to avoid misunderstanding, the system should paraphrase the user's input: '11/12/81' might be acknowledged by '11. Dec. 1981' in one country, and by 'Nov 12, 1981' in another; in either case the user is likely to notice any misinterpretation and correct it.

One last admonishment: clear and concise formulation of instructions is hard, but is well worth careful thought on the part of the author of a dialog. The colloquial style that makes human communication versatile is ill-advised in man-machine dialogs because it is hardly ever backed up by the intelligence necessary to make informal conversation effective. 'Would you like an opportunity to practice?' sounds fine until you discover that the program does not know how to react to the natural reply *What do you have to offer?*

As a general rule computer dialogs should present options rather than ask questions. There are two objections to computer dialogs based on questions: A question contains little information and it invites answers that may be beyond the program's ability to understand. Do you want to exit? leaves me in doubt as to what system state will occur if I answer Y, yes, no, or what else can I do?. The instructions:

```
Press X to exit to the top command level
      R to repeat the exercise
      L for a list of other options
```

may take three times as much space on the screen, but contain six "chunks of information" instead of the single one in the original question. Whereas the question only told me that I had the option to exit, without telling me where I would end up, the menu specifies for each of three commands how it is invoked and where it brings me. *Say it, don't ask* is a useful rule for finding unambiguous ways of expressing your message.

In striving for clarity and efficiency, the author of a dialog must also resist the temptation to play cute games. I've often used a formatting program called Sally. Sally had the disconcerting habit of introducing herself with a random name. *I'm Jennifer, what can I do for you?* I now take it for granted that any time a woman's name appears on the screen I must be in the formatter. As a novice I wasted time discovering Jennifer's text formatting skills.

Our attempt at classifying errors included parts of a solution to the problem of how to avoid them, in the form of a few general principles and many isolated suggestions for improvement. There remains the task of integrating these insights into a design technique. In the following sections we sketch our approach to this problem, first with respect to the system's behavior, then to its implementation.

3. An interactive system as seen by the user

Observation of casual users at a terminal provides valuable insight into the fundamental design question of how a machine should present itself. Most of the recurring difficulties they encounter are characterized well by the following questions:

- **where am I?** (when the screen looks different from what he expected)
- **what can I do here?** (when he is unsure about what commands are active)
- **how did I get here?** (when he suspects having pressed some wrong keys)
- **Where else can I go and how do I get there?** (when he wants to explore the system's capabilities).

We are beginning to learn that the logical design of an interactive system must allow the user to obtain a convenient answer to the questions above **at all times.** In other words, the man-machine interface must include queries about the state of the system (without changing this state), about the history of the dialog, and about possible futures. This principle is much more important for today's computerized machines, *black boxes* that show the user only as much about their inner working as the programmer decided to show, than it is for mechanical machines of the previous generation, which by visible parts, motion and noise continuously give the user a lot of state information.

In order to answer the user's basic questions in a systematic way, the designer of an interactive system must also design a simple **user's model of the system**: a structure that explains the main concepts that the user needs to understand and relates them to each other. The major concepts will certainly include many from the following list: the *types of objects* that the user has to deal with, such as files, records, pictures, lines, characters; *referencing mechanisms*, such as naming, pointing; *organizational structures* used to relate objects to each other, such as sets, sequences, hierarchies, networks; *operations* available on various objects; *views* defined on various types of objects, such as formatted or unformatted text; *commands* used to invoke operations; the *mapping of logical commands onto physical I/O devices*; the *past dialog*, how to store, edit, and replay it; the *future dialog*, or help facility.

The list is long and the design of the user's model of the system is an arduous task; but it should be possible to explain the overall structure of the model in half an hour. If the system's behavior then constantly reinforces the user's understanding of this structure, it will quickly become second nature to him. In contrast, if the system keeps surprising the user, it interferes with his memorization process and slows it down. Because questions will always arise during the learning phase, and because the right manual is rarely at the right place at the right time, an interactive system should be self-explanatory: it must explain the user's model of the system to the user, at least in the form of an on-line manual, preferably in the form of an integrated help facility that gives information about the user's current data and command environments at the press of a single key.

The user's model of the system is a *state-machine*: it has an *internal state* and an *input-output behavior.* Components of the state must include:

- **the user's data environment** (data currently accessible)
- **the user's command environment** (commands currently active).

The questions *Where am I?* and *What can I do here?* are then answered by displaying the current data and command environments, respectively [NW 80]. The user must be able to invoke this system display at all times, regardless of which applications program he is currently in, and its presentation must not change the state of the system in any way.

The system must have *universal commands*, which are active at all times. At least the state inquiry commands postulated above must be universal. In a highly integrated system many more commands can be made universal. General dialog commands that are needed in every interactive utility are taken out of the text editor, the diagram editor, the data base query language, and incorporated as universal commands into the system. The consequence on the user's view of the system is that all these utilities "talk the same language", and that in order to become proficient at using a new editor he only has to learn a small number of new commands: the data-specific commands are new, the data-independent commands remain the same.

4. Programming the man-machine interface

We have discussed the implications of the principles explained in this paper for the system's behavioral design. For a discussion of the implications for a system's internal structure I refer to other publications that present experience gained from implementing prototypes based on the ideas explained above. In any case the consequences must be discussed separately for the case of writing interactive application programs on an existing system, and for the design and implementation of an integrated interactive system. [Nie 80] *A pragmatic introductionto courseware design* is a tutorial for the first case. [XS 82] describes the goals, the design, and the implementation of an experimental interactive system that attempts to live up to all the principles stated in this article.

Most programmers have traditionally judged the quality of their programs from the point of view of machine efficiency and of internal structure. It is satisfying to observe that many have recently also become concerned with the quality of the man-machine dialog that their interactive programs conduct, as judged from the user's point of view. Well they might, for the computer user population is changing rapidly. Until recently only a small minority of computer users had the opportunity to use interactive systems. They were mostly professionals who use the computer daily: they are concerned with the inherent power of the computer for their application and get so used to its idiosyncrasies that a mysterious or even illogical man-machine interface doesn't bother them any more. An example of this situation is provided by UNIX, widely acclaimed as a model of a well-designed operationg system. [Nor 81] reaches the conclusion "the system design is elegant but the user interface is not". With the spread of low-cost single-user computers many casual and occasional users join the professionals, and for them the quality of the man-machine interface is crucial - an interactive system is only useful if the learning effort is commensurate with the brevity of the task they want to accomplish. It is for their benefit that computer professionals should start paying as much attention to their communicative skills as writers have always done.

Acknowledgments

My interest in man-machine communication was awakened while working on the PLATO project at the University of Illinois, to which I owe many insights. The experimental systems XS-0 and XS-1 served as test beds for evaluating the design techniques presented in this paper. I am indebted to my co-workers on these projects, in particular to G. Beretta, H. Burkhart, P. Fink, B. Plattner, J. Stelovsky, H. Sugaya, A. Ventura, and J.Weydert. This is a revised version of a paper presented at the 1982 International Zurich Seminar on Digital Communications, MAN-MACHINE INTERACTION, March 9-11, 1982.

References

[AP 79] *Apple Pascal Reference Manual*, APPLE COMPUTER INC., Cupertino, California, 1979.

[GV 80] E. Grandjean, and E. Vigliani (editors), *Ergonomic aspects of visual display terminals*, Proc. Intern. Workshop Milano, March 1980, Taylor and Francis Ltd., London, 1980.

[Fol 80] J. D. Foley, *The structure of interactive command languages*, in R. A. Guedj (ed.), Methodology of Interaction, 227-234, North Holland Publ. Co., Amsterdam 1980.

[HBR 81] P. Hayes, E. Ball, and R. Reddy, *Breaking the man-machine communication barrier*, IEEE Computer, Vol 14, No 3, 19-30, March 1981.

[LWSS 80] H. F. Ledgard, J. A. Whiteside, W. Seymour, and A. Singer, *An experiment on human engineering of interactive software*, IEEE Trans. Software Engr., Vol 6, No 6, 602-604, Nov 1980.

[Mil 56] G. A. Miller, *The magical number seven, plus or minus two: Some limits on our capacity for processing information*, Psych. Review, Vol 63, No 2, 81-96, March 1956.

[Mor 81] T. P. Moran (guest editor), *The psychology of human-computer interaction*, Special Issue, ACM Computing Surveys, Vol 13, No 1, March 1981.

[NW 80] J. Nievergelt and J. Weydert, *Sites, modes, and trails: Telling the user of an interactive system where he is, what he can do, and how to get places,*
in R. A. Guedj (ed.), Methodology of Interaction, 327-338, North Holland Publ. Co., Amsterdam 1980.

[Nie 80] J. Nievergelt, *A pragmatic introduction to courseware design,*
IEEE Computer, Vol 13, No 9, 7-21, Sep 1980.

[Nie 82] J. Nievergelt, *The computer-driven screen: an emerging two-way mass communications medium,*
Educational Media International, Int. Council for Educational Media, London, 7-12, No 1, 1982.

[Nor 81] D. A. Norman, *The trouble with UNIX,* 139-150, Datamation, Nov 1981.

[Nor 82] D. A. Norman, *Steps toward a cognitive engineering: Design rules based on analysis of human errors,* Proc. Conf. on Human Factors in Computer Sysyems, Gaithersburg MD, Mar 1982.

[XS 82] G. Beretta, H. Burkhart, P. Fink, J. Nievergelt, J. Stelovsky, H. Sugaya, J. Weydert, A. Ventura, *XS-1: An integrated interactive system and its kernel,*
Proc. 6-th International Conference on Software Engineering, Tokyo, 1982.